Procesadores de Lenguajes

Nuestro lenguaje

María Arranz Lobo
Celia Rubio Madrigal

mayo de 2021

1 Parte requerida

1.1 Cambios desde la última entrega

Se han añadido punteros y, por tanto, memoria en el montón.

Se han modificado ligeramente las prioridades de los operadores, así como añadidos algunos para los punteros.

Se ha hecho la parte de generación de código.

1.2 Partes no implementadas

En la generación de código, no se han implementado las siguientes características:

- El bucle for. Se haría haciendo una reserva para la copia de los elementos, o de sus referencias, y accediendo a ellas dentro de su ámbito.
- La inicialización del array mediante el par longitud, valor inicial. Se haría similar a la inicialización directa en modo lista.
- Los registros. Sus campos se guardarían y se accedería a ellos mediante su posición dentro de la declaración.
- La función `length()`.
- Las variables globales. Se tendrían que inicializar al comienzo del main, y hacer una comprobación sobre su contexto cada vez que se llamaran.

- Asignación de arrays enteros salvo al inicializar.
- Imports totales con $*$.
- Algunas funcionalidades de & como cambios de contexto.

La parte de mover arrays es bastante poco ortogonal, y se podría hacer mucho más limpio con las funcionalidades aisladas.

2 Tipos y expresiones

2.1 Tipos básicos

Tenemos tres tipos básicos: enteros `ent`, buleanos `bul` y caracteres `car`. Son los únicos tipos que se pueden leer por la entrada, usando la función reservada `kin()`, e imprimir, mediante la instruccion `kut` (ver instrucciones).

2.2 Tipos compuestos

Tenemos dos tipos compuestos: el tipo array `arr`, y el tipo puntero $*$.

2.2.1 Arrays

El tipo array tiene la siguiente sintaxis:

```
arr\ent v1 = [1,2,3].
arr\ent v2 = (3,0).
```

Se pueden inicializar o con una lista de valores, o con un par (longitud, valor inicial para todo elemento). Se accede a sus elementos mediante el operador `!` (ver operadores). Los índices comienzan en 0.

Tienen longitud fija, que se consulta con la función reservada `length()`, que admite el array a consultar como parámetro y devuelve un entero.

```
arr\arr\ent m1 = [[1,1,1],v1,v2].
arr\arr\ent m2 = (3,[0,0,0]).
arr\arr\ent m3 = (3,(3,0)).
```

2.2.2 Punteros

El tipo puntero tiene la siguiente sintaxis:

```
* ent a = new ent (1).
ent b = *(new ent (1)) + 2 + *a.
* ent c = &b.
```

Las expresiones de tipo puntero pueden formarse o con un `new`, que asigna memoria del montón del tamaño (constante) expresado entre paréntesis, o mediante una referencia a una expresión designable ya creada (el operador &).

2.3 Tipos del usuario

También cuentan como tipos los identificadores definidos durante la definición de registros, y los definidos durante la definición de alias.

2.4 Expresiones

Las expresiones pueden ser o básicas o formadas por operadores.

2.4.1 Expresiones básicas

Para el tipo `ent` las constantes son números enteros positivos.

Para el tipo `bul` las constantes son `si` y `no`.

Para el tipo `car` son un caracter rodeado por comillas simples: `'c'`.

Las expresiones básicas también pueden ser:

- Un identificador definido anteriormente en su mismo ámbito. Los identificadores solo pueden tener letras minúsculas, números y barras bajas (ambos no al comienzo). Se prefiere el estilo de *snake_case*.
- Una expresión con `new`.
- Una expresión rodeada por paréntesis,
- Una llamada a función.
- Una de las dos expresiones de creación de array.

2.4.2 Operadores

Op.	Tipo	Prioridad	Asociatividad	Semántica
|	Binario infijo	0	Izquierda	Or
&	Binario infijo	0	Izquierda	And
!=	Binario infijo	1	Izquierda	Distinto
==	Binario infijo	1	Izquierda	Igual
<	Binario infijo	2	Izquierda	Menor
<=	Binario infijo	2	Izquierda	Menor o igual
>	Binario infijo	2	Izquierda	Mayor
>=	Binario infijo	2	Izquierda	Mayor o igual
¬	Unario prefijo	3	No	Not
+	Binario infijo	3	Izquierda	Suma
−	Binario infijo	3	Izquierda	Resta
∗	Binario infijo	4	Izquierda	Multiplicación
/	Binario infijo	4	Izquierda	División entera
%	Binario infijo	4	Izquierda	Módulo
−	Unario prefijo	5	No	Opuesto
!	Binario infijo	6	Izquierda	Índice array
−>	Binario infijo	7	Izquierda	Acceso registros
∗	Unario prefijo	8	Sí	Puntero
&	Unario prefijo	8	Sí	Referencia

3 Estructura

La estructura de un fichero de nuestro lenguaje es la siguiente, en este orden:

- Una lista (potencialmente vacía) de imports.
- Una lista (potencialmente vacía) de declaraciones: de variables, de variables constantes, de registros, de definición de funciones, de definición de registros, o de definición de tipos (alias).
- Un main.

Los comentarios son con el símbolo ∈ hasta el final de línea.

3.1 Imports

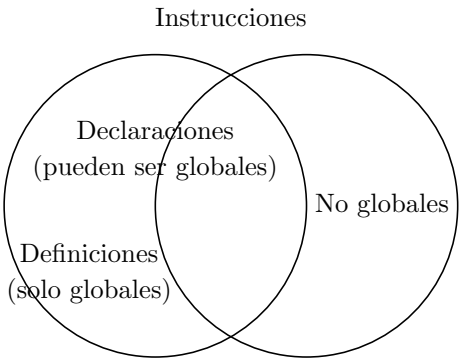
Los imports tienen la siguiente sintaxis:

```
import funcion from archivo.
import * from archivo.
```

Donde `archivo` es un fichero en la misma carpeta que el fichero raíz, y la definición de `funcion` está en la parte de declaraciones de ese fichero. Si se toma la segunda fórmula, se importan todas las declaraciones globales.

3.2 Declaraciones

Las declaraciones son un tipo concreto de instrucción que puede colocarse en el ámbito global. El resto de instrucciones no pueden. Además, las declaraciones que son definiciones no pueden usarse en otros bloques de instrucciones que no sean el global. Resumiendo:



Las definiciones se sobreescriben por orden si se repiten en el mismo ámbito.

3.2.1 Variables

La declaración de tipos básicos y compuestos siempre se hará con valor inicial. Fuera del `main` son globales, pero pueden ir en otros ámbitos. Además, pueden definirse como constantes.

```
const ent taxicab_number = 1729.
```

3.2.2 Definición de registros

Los campos del registro se definen mediante una lista de declaraciones de variables o de registros. Tienen la siguiente sintaxis:

```
data coleccion1 () = {
  arr\ent cosas = [0].
  bul vacio = no.
} .
```

Pueden llevar parámetros de inicialización entre paréntesis, pero deberán ser constantes. Eso sí, pueden ser de tipo puntero.

```
data coleccion2 (const ent size) = {
  arr\ent cosas = (size, 0).
  bul vacio = size==0.
} .
```

3.2.3 Registros

Fuera del `main` son globales, pero pueden ir en otros ámbitos. Se accede a sus componentes mediante el operador $->$.

```
coleccion1 bolsa1().
bolsa1->cosas ! 0 = 1.

coleccion2 bolsa2(3).
bolsa2->cosas ! 2 = 1.
```

3.2.4 Definición de funciones

No hay restricción sobre los tipos de los parámetros, pero el tipo del resultado no puede ser un array o un registro.

```
function copy return ent (arr\ent v, *arr\ent w) {
    *w = v.
    return 0.
}
```

3.2.5 Definición de tipos

Es la definición de un alias para un tipo que ya exista.

```
type matriz = arr\arr\ent.
```

4 Sección main

Es una secuencia de instrucciones rodeada por `main{ *** }` por donde se empieza a ejecutar el programa.

4.1 Instrucciones

Todas las instrucciones acaban en punto, salvo la definición de funciones que ya hemos descrito, el while, el for, el if y el if else.

4.1.1 Asignación

Solo se pueden asignar expresiones que sean un identificador, un acceso a array, un acceso a registro o un acceso a puntero. La sintaxis es:

```
my_num = 0 .
my_arr ! 2 = 1 .
my_data -> field = 2 .
*my_p = 3 .
```

4.1.2 Llamada a función

Además de usarse como una expresión, las llamadas a funciones pueden realizarse como una instrucción donde se descarta el resultado devuelto.

```
*arr\ent w = new arr\ent.
ent res = copy(v1,w) .
copy(v1,w) .
```

4.1.3 While

Se repite el bloque de instrucciones entre corchetes hasta que se deje de cumplir el valor del booleano entre paréntesis.

```
while(b) {
    b = ¬b.
}
```

4.1.4 For

Se recorren los elementos de un array dado en modo lectura.

```
arr\ent lista = [1,2,3].
for(l : lista) {
    kut(l).
}
```

4.1.5 If, if else

La definición de ramas de ejecución condicional se hará a través de las estructuras if (1 rama) o if-else (2 ramas). De manera que al tener la posibilidad de anidamiento permitirán definir la totalidad de casos que sea necesario.

```
if (num < 0) {
    num = -1 * num.
}

if (num % 2==0) {
    par = si.
}
else {
    par = no.
}
```

4.1.6 Impresión

Solamente de tipos básicos. Su sintaxis es:

```
kut(expresion).
```

4.1.7 Return

Dentro de una definición de función, para la ejecución de dicha función y devuelve el valor de su expresión. En el main para la ejecución total y devuelve un entero. Su sintaxis es:

```
return expresion.
```